The relation between diamond tiling and hexagonal tiling

Tobias Grosser
INRIA and École Normale Supérieure, Paris
tobias.grosser@inria.fr

Sven Verdoolaege
INRIA, École Normale Supérieure and KU Leuven
sven.verdoolaege@inria.fr

Albert Cohen
INRIA and École Normale Supérieure, Paris
albert.cohen@inria.fr

P. Sadayappan
Ohio State University
saday@cse.ohio-state.edu

ABSTRACT

Iterative stencil computations are important in scientific computing and more and more also in the embedded and mobile domain. Recent publications have shown that tiling schemes that ensure concurrent start provide efficient ways to execute these kernels. Diamond tiling and hybrid-hexagonal tiling are two successful tiling schemes that enable concurrent start. Both have different advantages: diamond tiling is integrated in a general purpose optimization framework and uses a cost function to choose among tiling hyperplanes, whereas the more flexible tile sizes of hybrid-hexagonal tiling have proven to be effective for the generation of GPU code.

We show that these two approaches are even more interesting when combined. We revisit the formalization of diamond and hexagonal tiling, present the effects of tile size and wavefront choices on tile-level parallelism, and formulate constraints for optimal diamond tile shapes. We then extend the diamond tiling formulation into a hexagonal tiling one, combining the benefits of both. The paper closes with an outlook of hexagonal tiling in higher dimensional spaces, an important generalization suitable for massively parallel architectures.

1. INTRODUCTION

Stencil computations are an important computational pattern in both scientific and engineering applications and they are becoming increasingly important in the embedded and mobile domain. Computational electrodynamics [13] or partial differential equations [11] are common use cases of stencils in high performance computing, whereas image and video processing are about to become driving forces in the embedded market. Even though manual and automatic optimizations of stencil computations have been designed since many years, the generation of efficient code remains a challenge especially for higher-dimensional stencils or for platforms which allow highly parallel execution on different hardware levels. With the increased use of parallel hardware in mobile markets as well as the foreseeable increase of three dimensional processing in upcoming embedded devices, a need emerges for solutions that facilitate the automatic generation of high-performance stencil codes for different devices.

For stencil computations, the tiling strategies that enable reuse along the time dimension have shown to be most efficient. Unfortunately, the standard approach uses parallel wavefronts in a skewed index space. These skewed wavefronts reduce tile-level parallelism [9] and induce load-imbalanced prologue and epilogue phases. Split tiling [5, 9] and overlapped tiling [8, 9] address this problem by enabling concurrent start along one of the original iteration space dimension. In other words, the tile schedule allows a wavefront of tiles parallel to one of the original dimensions of the index space to be executed in parallel. However, these two tiling techniques require either periodically alternating tile shapes or induce redundant computations. In contrast, the recently published diamond tiling [2] and hybrid-hexagonal tiling [6] schemes successfully obtain concurrent start without the need for redundant computations or multiple tile shapes.

Diamond tiling is a tiling strategy that uses a single n-dimensional paralleloptope[1] that is calculated such that it is possible to create a tiling that ensures that the number of tiles executable in parallel remains consistent throughout the computation, meaning that the tile schedule enables concurrent start. The advantages of diamond tiling are its integration in a general purpose compilation framework and the use of an adaptable cost function to determine tile shapes. Hybrid hexagonal-classical tiling is a tiling scheme that uses hexagonal tile shapes to enable concurrent start and to provide flexible tile size choices on one dimension. On the remaining dimensions it uses classical parallelogram tiling. The more domain specific formulation of hybrid-hexagonal tiling does not optimize tile shapes for a certain cost function, but always uses the most narrow dependence cone to derive the tile shape. On the other side, hybrid-hexagonal tiling has the advantage that it allows to adjust the time-tile height and the width along the space dimension individually. It also permits the creation of tiles with a flat summit and can ensure that tiles do not only have the same rational shape, but their integer point placement is by construction identical—all properties that have shown to be essential for efficient GPU code generation. Besides these advantages, there are also open problems. Even though diamond tiling

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://www.exastencils.org/histencils/2014/

[1] A parallelotope is a general term for what is known in 2D as parallelogram and in 3D as parallelepiped.

generally explains how to derive tiling hyperplanes that enable concurrent start, a tile schedule that includes both the tile sizes as well as the parallel wavefront coefficients necessary to obtain concurrent start was not presented. Hexagonal tiling has shown beneficial for higher dimensional stencils when combined with other tiling schemes, but the formulation of hexagonal tiling itself is limited to the 2D case (1 time dimension, 1 space dimension).

This paper combines the two tiling strategies to get the best of both worlds. Its contributions are: a) an in-depth analysis of the constraints that diamond-tiling imposes on tile-sizes and wavefront coefficients, b) a formulation of conditions that ensure identical placement of integer points within the tiles, c) an extension of the original diamond tiling algorithm to a hexagonal tiling algorithm for 2 dimensional problems (1 time dimension, 1 space dimension), d) ideas for hexagonal tiling of higher dimensional stencils.

The paper is structured as follows. In Section 2 we revisit diamond tiling, provide insights on tile size and wavefront coefficient constraints and give conditions that ensure important properties of the diamond tiles. We then introduce the unified hexagonal tiling scheme in Section 3 which includes a full formulation for two dimensional tiling as well as an outlook on hexagonal tiling for higher-dimensional cases. We discuss related work in Section 4 and conclude in Section 5.

2. DIAMOND TILING

Diamond tiling [2] is a tiling technique for stencil computations where the main contribution is the combination of affine transformations and a rectangular tiling that enables concurrent start. The idea of concurrent start is to ensure that the wavefront of tiles that are executed in parallel is aligned to a concurrent start hyperplane (normally an iteration space boundary) such that the number of tiles that are executed in parallel remains constant throughout the entire computation. This ensures that already at the beginning of the computation a sufficient amount of parallelism is available. Even though the name "diamond" suggests that the tile shapes are rhombi or rhombohedra (a.k.a. diamonds) and Figure 12 in Bandishti et al. [2] also uses edges of identical length, the tile shapes formed by diamond tiling are not restricted to diamonds, but can be more general parallelograms (parallelotopes in higher dimensions) as can be seen in Figure 3 and Figure 9a. However, some restrictions to the tile shape and sizes must be enforce to ensure that concurrent start is possible.

2.1 The Pluto optimizer

Diamond tiling was presented and implemented as an extension to Pluto [3], a general-purpose optimizer for data locality and parallelism. In contrast to other approaches that directly tile the iteration space (e.g., [5, 6]), the original Pluto tiling as well as diamond tiling are implemented as a two phase process. As a first step a program transformation is calculated that exposes sequences of loops (bands) that are tileable with rectangular tiles. In the second step a rectangular tiling is performed on these bands. Combined, this yields tiles with a possibly not rectangular, but parallelotope tile shape. There are several benefits of separating these two concerns. First, when calculating the parallel bands Pluto can and does perform other optimizations, e.g., data locality optimizations such as loop fusion. Second, tiling of the transformed program makes the tile shapes independent of the tiling hyperplanes, which makes the tiling easier to describe and analyze.

Pluto calculates program transformations on a polyhedral representation. In this representation the set of executed program statements (the iteration space) is modeled with a multi-dimensional integer set where each element represents an individual statement iteration. The execution order of elements of the iteration space is described by the schedule, an integer map that assigns a possibly multi-dimensional relative execution time to each element of the iteration space. Program transformations are performed by modifying the schedule. For a single statement and a k-dimensional execution time such a schedule has the form $S = \mathbf{x} \to (\mathbf{h}_0 \cdot \mathbf{x}, \ldots, \mathbf{h}_k \cdot \mathbf{x})$, where $\mathbf{x}$ is an element of the iteration space, $\mathbf{h}_i, i \in [0, k]$ are tiling hyperplanes and $\mathbf{h}_i \cdot \mathbf{x}$ denotes the sum of the per element products of $\mathbf{h}_i$ and $\mathbf{x}$. The result of Pluto's first step are exactly these tiling hyperplanes, selected such that the distance between two statements that depend on each other is not only lexicographically nonnegative (needed for validity of the schedule), but that the distance is also nonnegative at each individual dimension. For the exact algorithm on how to select such hyperplanes, we refer to [3]. For this paper, it is sufficient to understand that the all nonnegative dependence vectors make rectangular tiling valid. We present the Pluto rectangular tiling as a schedule only transformation which we believe is easier to understand than the actual Pluto transformation which modifies the iteration space as well. Conceptually, there should be no difference. Given a schedule S and a set of tile sizes $s_i, i \in [0, k]$ a rectangularly tiled schedule of S consists of two partial schedules. The first one, S_t, is placed at the outer level and enumerates the tiles itself. It is called the tile schedule. The second one, S_p, is placed at the inner level and enumerates the points within each tile. It is called point schedule. We define $S_t = (x_0, \ldots, x_k) \to (\lfloor(\mathbf{h}_0 \cdot \mathbf{x})/s_0\rfloor, \ldots, \lfloor(\mathbf{h}_k \cdot \mathbf{x})/s_k\rfloor)$ and $S_p = S$. This tiled schedule may already expose parallelism, but it may also be necessary to fall back to pipeline parallelism by forming a wavefront schedule at the outermost tile dimension. Then, such a wavefront schedule carries itself all dependences and ensures that the inner loops can be executed in parallel. This yields $S'_t = (x_0, \ldots, x_k) \to (\lambda_0\lfloor(\mathbf{h}_0 \cdot \mathbf{x})/s_0\rfloor + \cdots + \lambda_k\lfloor(\mathbf{h}_k \cdot \mathbf{x})/s_k\rfloor, \lfloor(\mathbf{h}_1 \cdot \mathbf{x})/s_1\rfloor, \ldots, (\mathbf{h}_k \cdot \mathbf{x})/s_k\rfloor)$ with $\lambda_i \in \mathbb{Z}_{\geq 0} : i \in [0, k]$. The coefficients λ_i allow the construction of different wavefronts. We call $\lambda_0 = \cdots = \lambda_k = 1$ the default wavefront coefficients. The hyperplanes that are calculated by the original Pluto algorithm allow the formation of such a wavefront schedule, but it is not always possible to form a tile schedule that is in the same direction as a given concurrent start face $\mathbf{f}$.

2.2 The diamond tiling extensions

Diamond tiling [2] extends the Pluto algorithm in a way that ensures that for the tiling hyperplanes computed there always exist wavefront coefficients that yield concurrent start. In the following, we identify a face or hyperplane to its orthogonal vector. This paper shows that "a transformation enables tilewise concurrent start along a face $\mathbf{f}$ if and only if the tile schedule is in the same direction as the face and carries all inter-tile dependences". It also shows that "concurrent start along a face $\mathbf{f}$ can be exposed by a set of hyperplanes if and only if $\mathbf{f}$ lies strictly inside the cone formed

by the hyperplanes, i.e., if and only if $\mathbf{f}$ is a strict conic combination of all the hyperplanes". This means it finds for a concurrent start hyperplane $\mathbf{f}$ tiling hyperplanes $\mathbf{h}_i$ such that the following equality holds:

$$m\mathbf{f} = \lambda_1\mathbf{h}_1 + \cdots + \lambda_k\mathbf{h}_k \quad (1)$$

$$\lambda_i, m \in \mathbb{Z}_{\geq 0}$$

The main focus of the diamond tiling paper is to prove the conditions necessary to ensure that the calculated hyperplanes can be used to construct a concurrent start schedule as well as to give an algorithm that actually calculates such hyperplanes. We consequently refer to this publication for details. One question that was explored less is under which conditions, especially for which tile sizes and for which wavefront coefficients, the rectangularly tiled schedule achieves concurrent start. Specifically, it is not clear for which values of λ_i, s_j the following holds:

$$m\mathbf{x}\mathbf{f} = \lambda_0\lfloor(\mathbf{h}_0\mathbf{x})/s_0\rfloor + \cdots + \lambda_k\lfloor(\mathbf{h}_k\mathbf{x})/s_k\rfloor \quad (2)$$

2.3 Relation between tile sizes and wavefronts

Even though the diamond tiling yields tiling hyperplanes that allow concurrent start, to construct the full tile schedule the tile sizes s_i as well as the wavefront coefficients λ_i still need to be chosen. Choosing the correct values is important, not only to ensure that the tiles executed within the wavefront are started concurrently, but also to control the horizontal distance between tiles of the same color relative to their tile size. We call this the density of the schedule, a property important to understand the amount of computation that can be performed in parallel. Before suggesting good values, we explore the impact of different choices.

Let us first consider a simple example with symmetric dependences:

```
for t
  for i
    A[t+1][i] = A[t][i-1] + A[t][i+1]
```

Pluto's diamond tiling implementation calculates for this kernel the transformation $(t, i) \rightarrow (t - i, t + i)$ and applies rectangular tiling in the transformed space. The default wavefront coefficients $\lambda_0 = \lambda_1 = 1$ are then used to enable parallel execution. This results in the tile schedule $(t, i) \rightarrow (\lfloor(t-i)/s_0\rfloor + \lfloor(t+i)/s_1\rfloor, \lfloor(t+i)/s_1\rfloor)$. The default square tile shapes ($s_0 = s_1$) yield both concurrent start as well as a high density of tiles. Figure 1 illustrates this for $s_0 = s_1 = 4$ with the tile wavefront highlighted in red and the concurrent start hyperplane highlighted in black. The two hyperplanes being parallel shows that the tile wavefront has concurrent start. When different tile sizes are chosen for the two dimensions the default wavefront no longer yields concurrent start. In Figure 2 we illustrate for $s_0 = 4, s_1 = 6$ that the default wavefront (red) is no longer parallel to the concurrent start hyperplane (black). It is possible to still get concurrent start using the non-default wavefront coefficients $\lambda_0 = 2, \lambda_1 = 3$, which yields the schedule $(t, i) \rightarrow (2\lfloor(t-i)/6\rfloor+3\lfloor(t+i)/4\rfloor, \lfloor(t+i)/4\rfloor)$. Unfortunately, a non-default wavefront causes a large loss in tile-level parallelism throughout the computation. This effect is illustrated by the yellow wavefront in Figure 2, which is parallel to the concurrent start hyperplane (black).

Next we analyze a kernel with asymmetric dependences:

```
for t
  for i
    A[t+1][i] = A[t][i-1] + A[t][i+2]
```

Pluto derives from this kernel the transformation $(t, i) \rightarrow (t - i, 2t + i)$. This transformation combined with square tiling and the default wavefront coefficients allows concurrent start as shown in Figure 3 for $s_0 = s_1 = 4$. The reason for this, possibly surprising, result is that for a 2 dimensional stencil (1 space, 1 time) with dependence distance 1 in the time direction, the coefficient of the space dimension in the normal will always be ± 1. This ensures that when adding the two hyperplanes together their coefficients for the space dimension cancel out and we get again the concurrent start hyperplane. Consequently, the default wavefront coefficients combined with square tile sizes yield a concurrent start wavefront. As already found earlier, non-square tile sizes will prevent concurrent start with the default wavefront coeffcients.

Another interesting observation is that even though the rational tile shapes in Figure 3 are identical throughout the original iteration space, the set of contained integer points is not. The reason for this difference is that even though we use integral tile sizes in the transformed space, the borders may become non-integral in the original space. Varying integer point placements between tiles can cause problems due to additional conditions in the generated code.

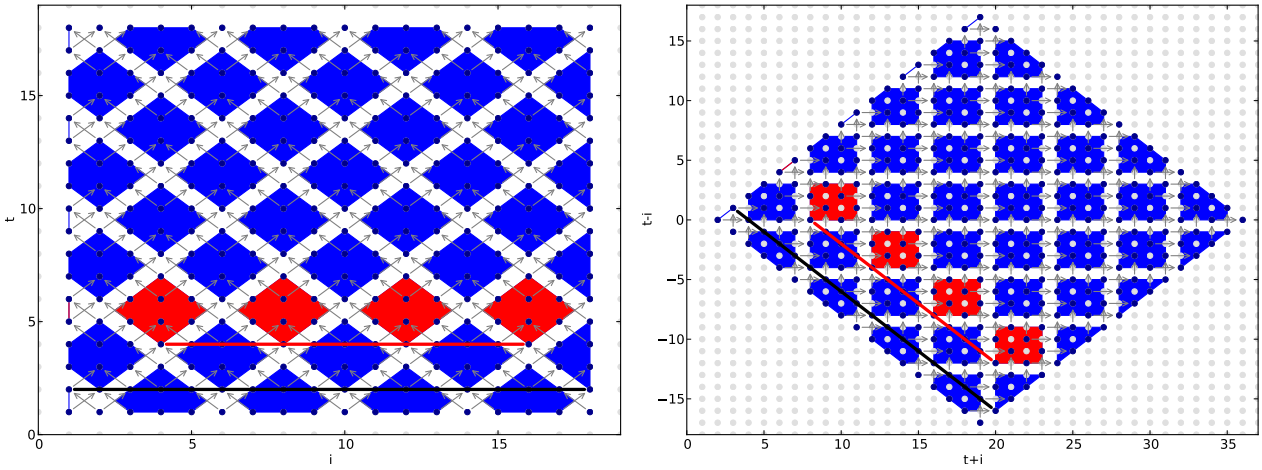
As a next step we look into a case that has dependence distances that have different lengths on the time dimension.

```
for t
  for i
    A[t+1][i] = A[t][i-1] + A[t-2][i+1]
```

For this kernel, the Pluto implementation derives the transformation $(t, i) \rightarrow (t + 3i, t + i)$. Note that this result is different from what the algorithm in [2] would produce. Apparently, the Pluto implementation is using a variation of that algorithm. It is not clear if there is a problem in this variation or that this is a mere implementation problem. As both hyperplanes have a positive coefficient for the space dimension, it is impossible to create a conic combination that eliminates the space dimension and yields a concurrent start hyperplane. According to the diamond tiling paper concurrent start is impossible and these are no valid diamond tiling hyperplanes.

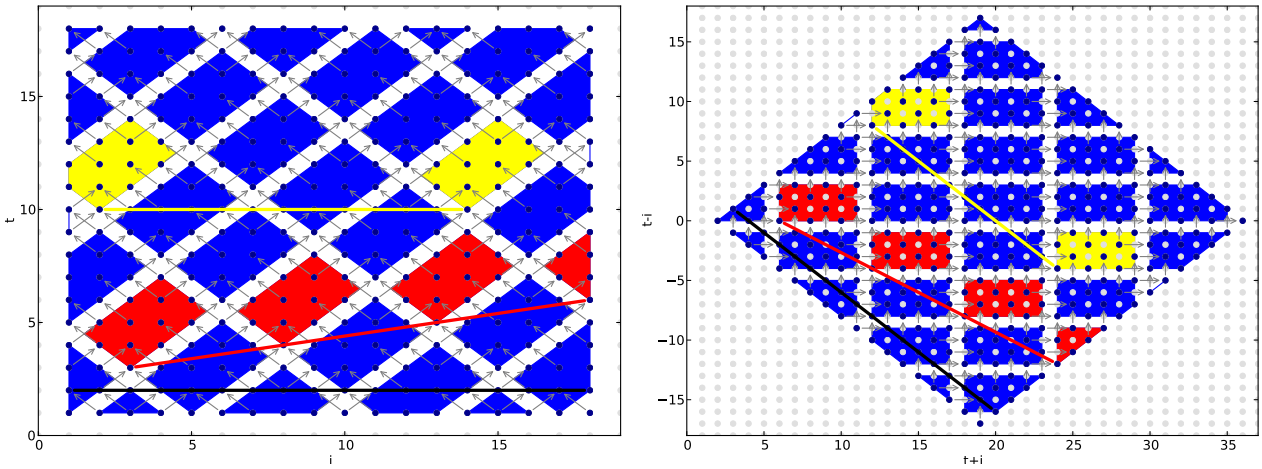
Even though the diamond tiling implementation in Pluto did not derive a valid tiling for the last kernel, there exist valid diamond tilings for it. One is the transformation $(t, i) \rightarrow (t - i, t + i)$. The same transformation was already chosen for the example illustrated in Figure 1 and according to our understanding of the cost function in Pluto, this is in fact the transformation that the algorithm of [2] would choose. The resulting tiling yields 8 computations for a per-tile memory footprint of 3.

Another valid diamond tiling transformation is $(t, i) \rightarrow (t + 3i, t - i)$. The hyperplanes in this transformation are the ones hybrid-hexagonal tiling would read off directly from the dependence cone. Given a different cost function, Pluto may also choose this transformation. The interesting point here is, that the normal of the concurrent start hyperplane in the transformed space is not anymore (1,1), but rather (1,3). In this case, the standard square tiling illustrated in Figure 4 only yields concurrent start if, instead of the default



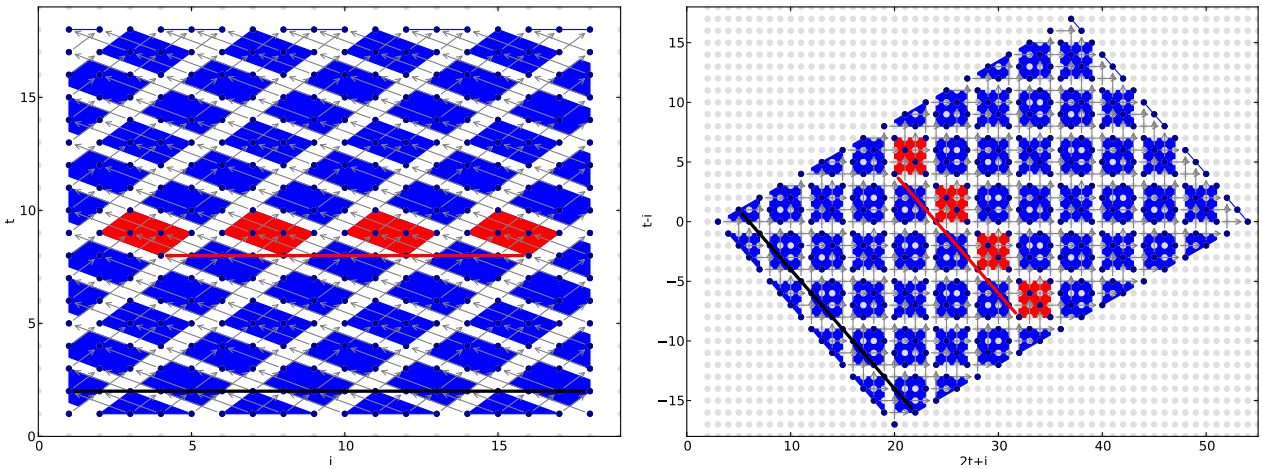
(a) original space (b) transformed space

Figure 1: Symmetric dependences & square tiling



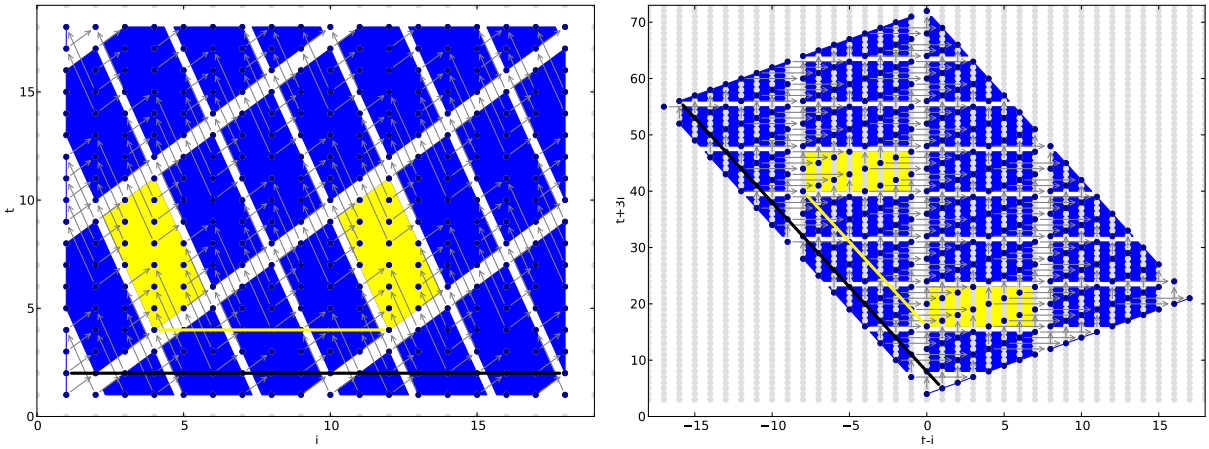
(a) original space (b) transformed space

Figure 2: Symmetric dependences & non-square tiling



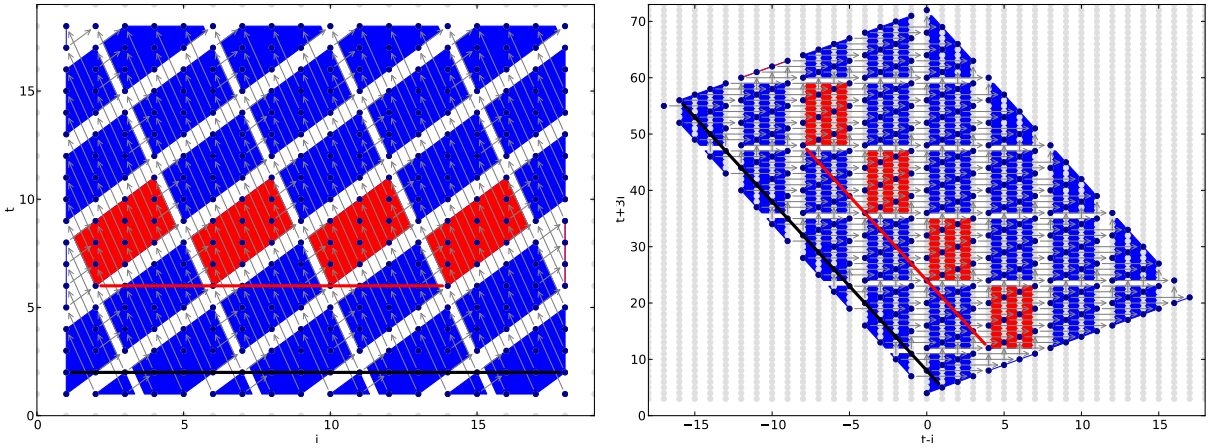
(a) original space (b) transformed space

Figure 3: Asymmetric dependences & square tiling



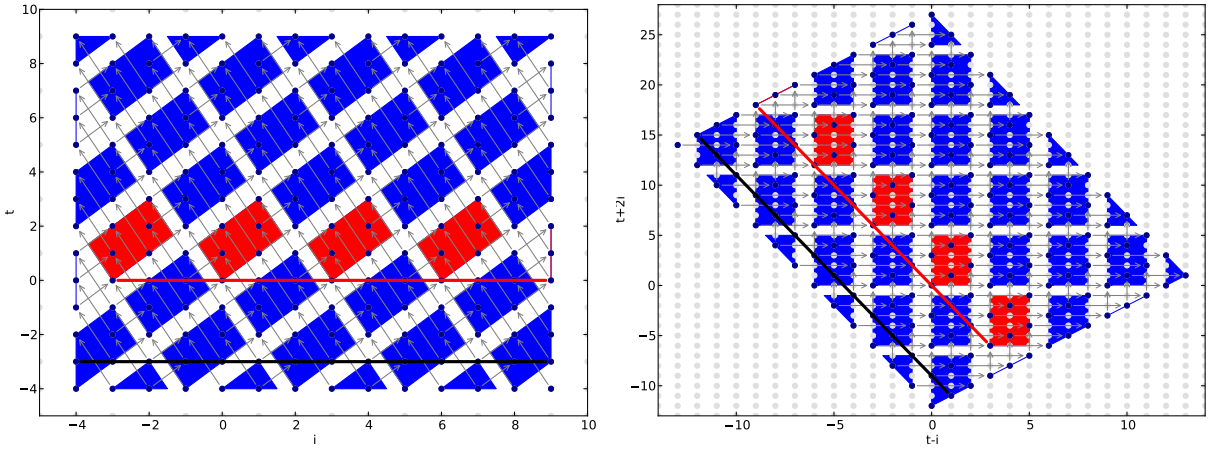
(a) original space (b) transformed space

Figure 4: More than one time step - Tiling read off from dependence cone and used by hexagonal tiling. Square tiles cause loss of tile-level parallelism.



(a) original space (b) transformed space

Figure 5: More than one time step - Tiling read off from dependence cone and used by hexagonal tiling. Non-square tiles ensure good efficiency and maximal tile-level parallelism.



(a) original space (b) transformed space

Figure 6: Diamond tiling

wavefront coefficients, $\lambda_0 = 1, \lambda_1 = 3$ are chosen. As shown earlier, this severely reduces tile-level parallelism. On the other hand, for the same memory footprint as before, this tiling executes 16 computations.

We can restore concurrent start with the default wavefront by using non-square tile sizes. Figure 5 shows a non-square tiling ($s_0 = 12, s_1 = 4$) which enables concurrent start, which has maximal tile-level parallelism and which reaches 12 computations for a memory footprint of three. Consequently, we would prefer this tiling over the previous two.

2.4 Optimal tiles with default wavefront

As seen in the previous section, the use of the default wavefront coefficients is necessary to ensure high tile-density. However, by itself it gives no guarantee neither for concurrent start nor does it ensure that all tiles share the same integer point placement. As those properties are important, we present the conditions under which they can be reached.

First, we explore the integer point placement. Assuming tiling hyperplanes $\mathbf{h}_i$ are combined into a matrix:

$$H = \begin{pmatrix} \mathbf{h}_0 \\ \vdots \\ \mathbf{h}_k \end{pmatrix}$$

then tile sizes that are multiples of the determinant of H will ensure that all tiles have the same configuration of integer points since $\det(H) \cdot H^{-1}$ is an integer matrix. The hyperplanes used, e.g., in Figure 3 yield

$$H = \begin{pmatrix} 1 & -1 \\ 2 & 1 \end{pmatrix}$$

and consequently $\det(H) = 1+2 = 3$. As $s_0 = s_1 = 4$ are not multiples of 3, the tiles differ in the integer point placement. For the same figure, tile sizes such as, e.g., $s_0 = s_1 = 3$ would ensure a uniform integer placement across all tiles. The above condition is sufficient independently of the chosen wavefront schedule.

Next, we investigate the conditions on tile sizes to ensure concurrent start with the default default wavefront coefficients. Let $h_{x,0}$ be the first component of $\mathbf{h}_x$ and $h_{x,1}$ the second. The default wavefront then is $\lfloor (h_{0,0}t + h_{0,1}i)/s_0 \rfloor + \lfloor (h_{1,0}t + h_{1,1}i)/s_1 \rfloor$. Now, to achieve concurrent start, we need to ensure that the default wavefront schedule only depends on the time dimension t and that all space dimensions (i.e., i) are eliminated. This is true under the condition $s_0/|h_{0,1}| = s_1/|h_{1,1}|$. Note that the wavefront may still depend on the fractional part of the space dimension, but this only results in a variation within a fixed range, independently of the size of the domain. We can see that in Figure 1, where we reach concurrent start for the default wavefront, this conditions holds with $4/1 = 4/1$. On the other hand, when changing the tile sizes to $s_0 = 4$ and $s_1 = 6$ as in Figure 2, the previous condition turns into $4/1 = 6/1$ and concurrent start is not possible with the default wavefront. The above shows that to obtain concurrent start the two tile sizes cannot be chosen independently, but need to be scaled together. To make this more clear we introduce a new variable s which can be chosen freely and which is then used to define $s_0 = s|h_{0,1}|$ and $s_1 = s|h_{1,1}|$ such that concurrent start is obtained.

3. UNIFIED DIAMOND AND HEXAGONAL TILING

In this section we present a extended formulation of diamond tiling which allows the creation of hexagonal tiles. The hexagonal tiles calculated are similar to the ones presented in [6], but are not identical in shape.

3.1 The schedule for hexagonal tiling (2D case)

Let us first consider a two-dimensional iteration space. To obtain such a schedule we start from the diamond tiling approach, which means we first calculate a set of tiling hyperplanes, transform the index space with these hyperplanes and then apply rectangular tiling in the transformed space. We then (optionally) transform the rectangular tiling by "stretching" the rectangular tiles along the concurrent start hyperplane. The stretched rectangular tiles in the transformed space form hexagonal tiles in the original space. As a result we have a single schedule that describes diamond tiling, if tiles are stretched by a vector of length zero, and hexagonal tiling, if they are stretched by a non-zero-length vector.

In the following description, we assume that the tiling hyperplanes h_0, h_1 are computed by the diamond tiling algorithm as described in [2]. We focus on the description of the (possibly) stretched tiling scheme in the transformed space. As input for the stretched tiling scheme, we take the tile sizes s_0, s_1 as well as a vector $\mathbf{v} = (v_0, v_1)$, which is parallel to the concurrent start hyperplane (in the transformed space). We also require that the direction vector of the concurrent start hyperplane $\mathbf{n} = (n_0, n_1)$ is strictly positive in all components, as guaranteed by the algorithm of [2].

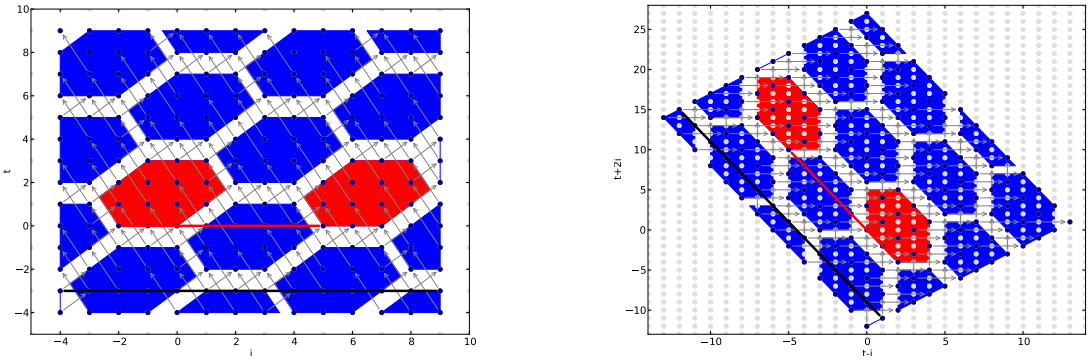
We first model diamond tiling using a standard 2D rectangular tiling in the transformed space. In this tiling the symbols s_0, s_1 define the tile sizes along the dimensions d_0, d_1 while T_0, T_1 are the resulting tile schedule dimensions (we ignore the point schedule dimensions, as this mapping is not interesting for this discussion). The following map describes such a rectangular tiling.

$$(d_0, d_1) \to (T_0, T_1) : s_0 T_0 \leq d_0 < s_0(T_0 + 1) \wedge s_1 T_1 \leq d_1 < s_1(T_1 + 1)$$

Our goal is to achieve and maintain concurrent start using the default wavefront. Consequently s_0 and s_1 cannot be chosen freely (see Section 2.4). We require the user to choose tile sizes that ensure concurrent start. Figure 6 illustrates the above rectangular tiling using the transformation $(t, i) \to (t + 2i, t - i)$, as well as the tile sizes $s_0 = 6, s_1 = 3$. The red tiles show the concurrent start wavefront.

Starting from this rectangular tiling we want to stretch the contained tiles by a vector $\mathbf{v}$ with components v_0, v_1, where $\mathbf{v}$ is parallel to the concurrent start hyperplane. In principle, $\mathbf{v}$ can have either of two possible directions, but to simplify the schedule formulation we choose $\mathbf{v}$ such that $v_0 < 0 \wedge v_1 > 0$. Figure 7 shows a stretching as we obtain it for $\mathbf{v} = (-4, 2)$ and $\mathbf{n} = (1, 2)$.

Before we implement the actual stretching, we first add two additional constraints to each tile. The first one bounds each tile at its lexicographic minimal point with the concurrent start hyperplane, the second one bounds each tile at its lexicographic maximal point with the same (but translated) hyperplane. We implement the lower boundary by placing the hyperplane at the origin and by offsetting it for each tile



(a) original space

(b) transformed space

Figure 7: Hexagonal-tiling

according to the tile sizes. To offset the tile along d_0 we adjust the right hand side of the lower bound by $n_0 s_0 T_0$ and $n_1 s_1 T_1$. The upper boundary is implemented by reversing the lower hyperplane. The location of the upper hyperplanes for tile (T_0, T_1) is the origin of tile $(T_0 + 1, T_1 + 1)$.

$$
\begin{aligned}
(d_0, d_1) \rightarrow (T_0, T_1) : \\
& s_0 T_0 \leq d_0 < s_0(T_0 + 1) \wedge \\
& s_1 T_1 \leq d_1 < s_1(T_1 + 1) \wedge \\
& n_0 s_0 T_0 + n_1 s_1 T_1 \leq n_0 d_0 + n_1 d_1 \wedge \\
& n_0 d_0 + n_1 d_1 < n_0 s_0(T_0 + 1) + n_1 s_1(T_1 + 1)
\end{aligned}
$$

As a last step, we now stretch the tiles along $\mathbf{v}$. This requires us to increase the size of the rectangular tiles by v_0 in the d_0 dimension and v_1 in the d_1 dimension. We also account for the shifted positions of the rectangular tiles by adding some offsets o_0, o_1 to the upper and lower tile boundaries that will be derived later in this section. Finally we adjust the locations of the concurrent start planes by using $c_0 = n_1(s_0 + v_0) + n_0 v_1$ and $c_1 = n_1(s_1 + v_1) + n_0 v_0$.

$$
\begin{aligned}
(d_0, d_1) \rightarrow (T_0, T_1) : \exists o_0, o_1 : \\
& o_0 = -v_0 T_0 + v_0 T_1 \wedge o_1 = -v_1 T_0 + v_1 T_1 \wedge \\
& s_0 T_0 + o_0 + v_0 \leq d_0 < s_0(T_0 + 1) + o_0 \wedge \\
& s_1 T_1 + o_1 \leq d_1 < s_1(T_1 + 1) + v_1 + o_1 \wedge \\
& c_0 T_0 + c_1 T_1 \leq n_0 d_0 + n_1 d_1 \wedge \\
& n_0 d_0 + n_1 d_1 < c_0(T_0 + 1) + c_1(T_1 + 1)
\end{aligned}
$$

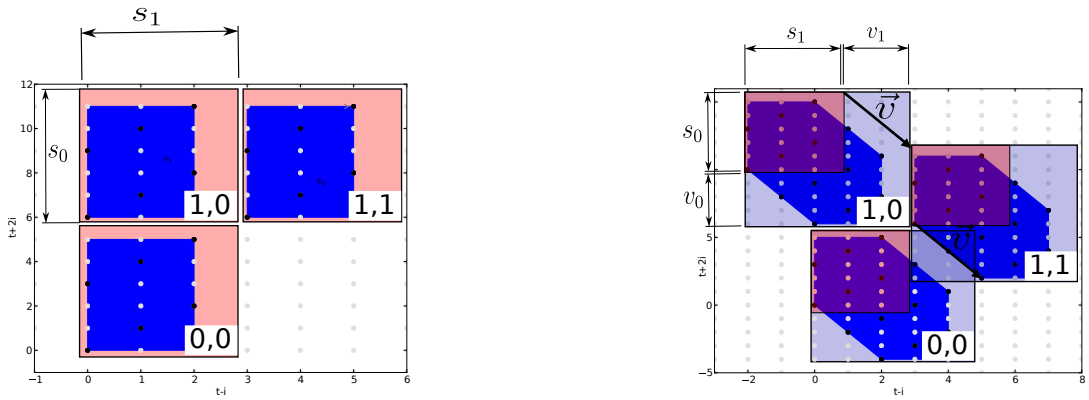
Figure 8 illustrates the last step in detail. On the left side we see in red the original square tiles (0,0), (1,0) and (1,1) each of size 6×4. On the right side, we see the tiles with the same tile numbers, but stretched along $\mathbf{v}$. We can see that the rectangular tile shapes have been extended by 4 along d_0 and by 2 along d_1 resulting in the light blue tile shapes (the dark blue tile shapes illustrate the contained integer points). We can also see that the position of the red tile shape of tile (0,0) has not moved. However, when going one step up to tile (1,0) which means increasing the tile number T_0 by one, we offset the tile by $-v_0$ along d_1 as well as $-v_1$ along d_1. Similarly, when going from tile (1,0) to tile (1,1) which means increasing the tile number T_1 by one, we offset the tile by v_0 along d_0 and v_1 along d_1. Combined this yields the offset $o_0 = -v_0 T_0 + v_0 T_1$ for d_0 and $o_1 = -v_1 T_0 + v_1 T_1$ for d_1. The new values c_0 and c_1 do now also take into account the offset of the plane. When varying T_0 we now do not only need to take the vertical tile size s_0 into account, but in addition we include the additional vertical offset v_0 as well as the changed horizontal offset v_1. To support concurrent start hyperplanes of different orientations such offsets are scaled by the relevant components of $\mathbf{n}$. The corresponding changes have been added when adjusting c_1.

A very important observation to make is that the tiles (T_0, T_1) as well as $(T_0 + 1, T_1 + 1)$ have overlapping rectangular tiles. However, the concurrent start hyperplanes that have been added right at the position of $\mathbf{v}$ ensure that the tiles are non-overlapping and still tile the full space. Also, as our stretching and translation was only along the concurrent start hyperplane, no dependences have been violated. Finally, if the previous tiling had concurrent start, stretching along the concurrent start hyperplane preserves this property.

3.2 Hexagonal tiling for higher dimensions

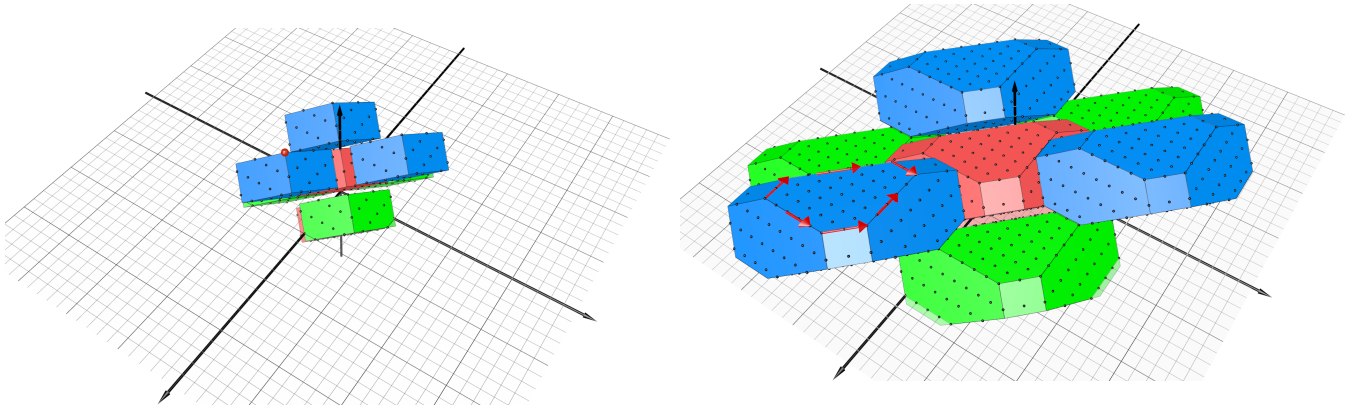
To extend our unified hexagonal tiling to higher dimensional kernels we use a shape derived from a truncated octahedra [4] to create a tiling for one time and two space dimensions, that not only provides two dimensions of parallelism, but that also gives the freedom to adjust the size of the tile shape independently for the different dimensions. Figure 9b illustrates such a tiling. In the illustration the time dimension goes upwards whereas the space dimensions go to the lower left and the lower right corner of the rendering. The hyperplane orthogonal to the time dimension is the concurrent start hyperplane. Tiles of the same color are executed at the same time step. As visible in the figure, the tiles of a single color are within a hyperplane parallel to the concurrent start hyperplane. The individual tiles of a single color are independent and can be executed in parallel. There is parallelism along both space dimensions. All tiles share a single tile shape.

The hexagonal tiling is derived from the diamond tiling illustrated in Figure 9a (the same example as used by Bandishti et al. [2]). At the beginning the peak of all tiles is formed by a single point (illustrated by the red dot on the lower left blue tile of Figure 9a). Similarly to the construction of hexagonal tiling for one space dimension, we then bound each tile at the top and at the bottom by the concurrent start hyperplane and stretch the peak to form a plane. However, for the case of two space dimension we "stretch" along three different vectors all chosen to be parallel to the concurrent start hyperplane and, in addition, to be inside one of the tiling hyperplanes. We illustrate in the blue



(a) unstretched (b) stretched

Figure 8: The stretching in the transformed space



(a) Plain Diamond tiling - unstretched

(b) Hexagonal tiling - Derived from the tiling in Figure 9a, but stretched along the concurrent start hyperplane

Figure 9: Hexagonal tiling - two space dimensions (3D rendering)

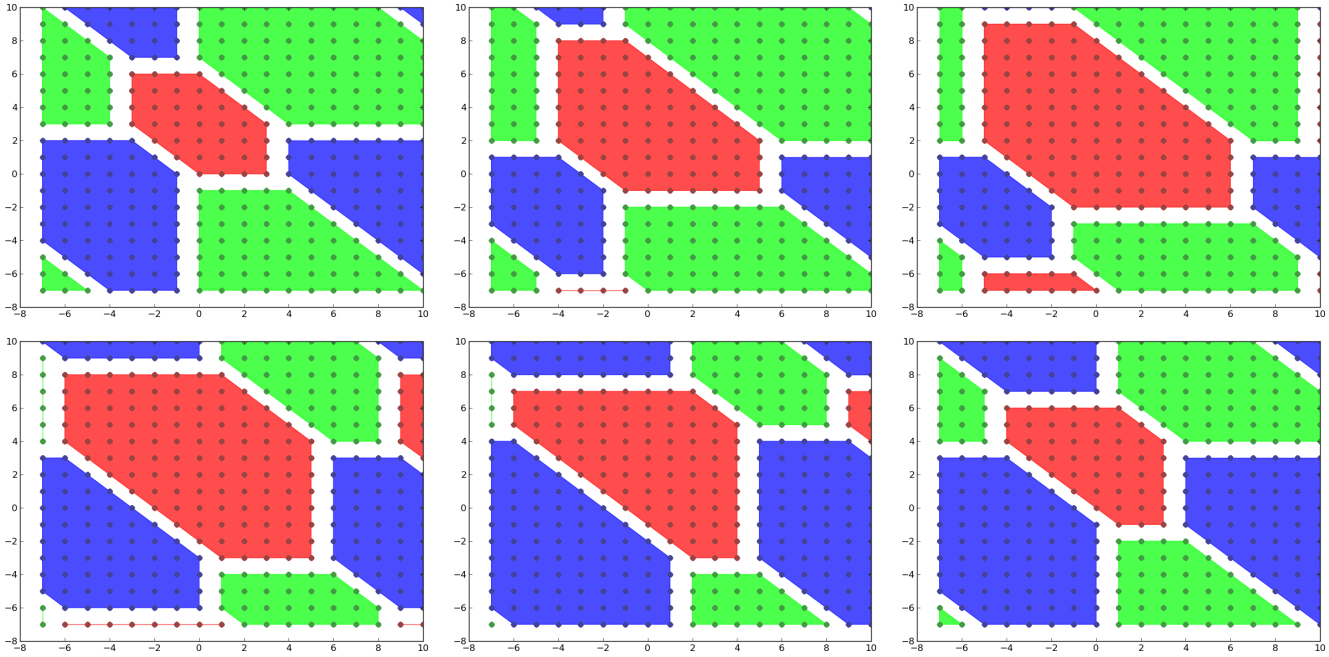


Figure 10: Hexagonal tiling - two space dimensions (Time steps 0-5)

tile at the lower left of Figure 9b these stretching vectors in red. Figure 10 illustrates that the tiling is space filling. By stretching only within the concurrent start hyperplane no dependences have been violated and concurrent start is preserved. The graphical illustration and the above claims only give an intuition of this tiling scheme. Additional work is required to understand the construction of such a tiling, its properties and its effectiveness. However, the promise we see is that we can translate the advantages of hexagonal tiling into higher dimensional cases — enabling flexible tile sizes, concurrent start as well as thread-level parallelism along multiple dimensions for higher-dimensional kernels.

4. RELATED WORK

Aside from the already discussed diamond and hybrid-hexagonal tiling [2, 6], there has been a lot of successful research in generating code to efficiently perform stencil computations. There is Pochoir [14], a domain-specific C++ framework as well as Henretty et al. [7] with a DSL-based approach. Strzodka [12] uses an in-tile wavefront traversal technique to achieve efficient cache use even with tile sizes larger than the available cache memory. All approaches generate efficient CPU code. Then, there are a set of general optimizers. PPCG [16] generates parallel CPU and GPU code using classical (time) tiling. It relies on affine transformations to extract parallelism and improve locality, using a variant of the Pluto algorithm [3]. Reservoir Labs' R-Stream is also a reference polyhedral compiler targeting GPUs [10, 15]. Par4All [1] is an open source parallelizing compiler developed by Silkan targeting multiple architectures. The compiler is not based on the polyhedral model, but uses abstract interpretation for array regions, performing powerful interprocedural analysis on the input code. Finally, there are tools that generate efficient GPU code. Here Holewinski's Overtile [8] and Grosser's split tiling [5] compilers represent, besides [6], the state-of-the-art for the automatic generation of efficient GPU code relying on overlapped and split tiling, respectively.

5. CONCLUSION

We presented a formulation of hexagonal tiling that combines the benefits of diamond tiling and hybrid-hexagonal tiling. Starting from the published diamond-tiling algorithm, we formulated conditions on tile sizes and wavefront coefficients to ensure concurrent start. We also formulated the condition that ensures the same integer point placement across all tiles. And most importantly, we extended the original diamond tiling algorithm to hexagonal tiles. The added flexibility of hexagonal tiles does not only make the choice of tile sizes more flexible but also enables the creation of tiles with a flat summit. Both these features have been shown useful for GPU code generation. Finally, we gave an outlook on our plans to extend this tiling scheme to higher dimensional stencils, an extension that will bring together flexible tile sizes and multiple dimensions of parallelism.

Acknowledgments. This work greatly benefited from regular discussions with Uday Bondhugula. It was partly funded by a Google European Fellowship in Efficient Computing, by the European FP7 project CARP id. 287767, the COPCAMS ARTEMIS project, and award 0926688 from the U.S. NSF.

6. REFERENCES

[1] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian McMahon, François-Xavier Pasquier, Grégoire Péan, Pierre Villalon, et al. Par4All: From convex array regions to heterogeneous computing. In *IMPACT*, 2012.

[2] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *ACM Supercomputing Conf.*, 2012.

[3] Uday Bondhugula, J. Ramanujam, and et al. PLuTo: A practical and fully automatic polyhedral program optimization system. In *PLDI*, 2008.

[4] H.S.M. Coxeter. Regular and semi-regular polytopes. i. *Mathematische Zeitschrift*, 46(1):380–407, 1940.

[5] Tobias Grosser, Albert Cohen, Paul HJ Kelly, J Ramanujam, P Sadayappan, and Sven Verdoolaege. Split tiling for gpus: automatic parallelization using trapezoidal tiles. In *GPGPU-6*. ACM, 2013.

[6] Tobias Grosser, Albert Cohen, Sven Verdoolaege, P. Sadayappan, and Justin Holewinski. Hybrid hexagonal/classical tiling for GPUs. In *International Symposium on Code Generation and Optimization*, 2014.

[7] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *ICS*. ACM, 2013.

[8] Justin Holewinski, Louis-Noël Pouchet, and P Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *ICS*, pages 311–320. ACM, 2012.

[9] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI*, pages 235–244, 2007.

[10] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *GPGPU-3*, 2010.

[11] G. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford University Press, 2004.

[12] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and H Seidel. Cache accurate time skewing in iterative stencil computations. In *Parallel Processing (ICPP), 2011 International Conference on*. IEEE, 2011.

[13] A. Taflove. *Computational electrodynamics: The Finite-difference time-domain method*. Artech House, 1995.

[14] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. The pochoir stencil compiler. In *SPAA*. ACM, 2011.

[15] Nicolas Vasilache, Benoit Meister, Muthu Baskaran, and Richard Lethin. Joint scheduling and layout optimization to enable multi-level vectorization. In *IMPACT*, Paris, France, January 2012.

[16] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM TACO*, 9(4):54, 2013.